MECHANISM TO REDUCE THE COST OF FORWARDING POINTER ALIASING

RELATED APPLICATION

[0001] This application claims the benefit of U.S. Provisional Application No. 60/299,244, filed on Jun. 19, 2001.

[0002] The entire teachings of the above application are incorporated herein by reference.

GOVERNMENT SUPPORT

[0003] The invention was supported, in whole or in part, by a grant F30602-98-1-0172 from Air Force Research Lab. The Government has certain rights in the invention.

BACKGROUND OF THE INVENTION

[0004] Forwarding pointers are an architectural mechanism that allow references to a memory location to be transparently forwarded to another location. Known variously as "invisible pointers,""forwarding pointers" and "memory forwarding," they are familiar to the hardware community but to date have been incorporated into very few architectures.

[0005] One reason that forwarding pointers have received little support is that they have been perceived as possessing limited utility. Recently, however, it has become apparent that forwarding pointers are indeed useful constructs that can expedite program execution. Chi-Keung Luk and Todd C. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation," Proc. ISCA 1999, pp. 88-99 (hereafter "Luk"), incorporated by reference herein in its entirety, show that using forwarding pointers to perform safe data relocation can result in significant performance gains on arbitrary programs written in C, speeding up some applications by more than a factor of two. Jeremy Brown, "Memory Management on a Massively Parallel Capability Architecture", Ph.D. thesis proposal, M.I.T., December 1999, gives an algorithm for performing asynchronous local compacting garbage collection in a massively parallel distributed system. This algorithm uses forwarding pointers to avoid the high run-time costs usually associated with such a system. Thus, there is growing motivation to include hardware support for forwarding pointers in novel architectures.

[0006] A second and perhaps more significant reason that forwarding pointers have received little attention from hardware designers is that they introduce aliasing—that is, it is possible for two different pointers to resolve to the same word in memory.

[0007] **FIG. 1A** illustrates how aliasing occurs. A first pointer P_2 **14** points directly to some target data **16**. A second, indirect pointer, P_1 **10** points to a forwarding pointer **12** which in turn points to the target data **16**. Thus, pointers P_1 and P_2, which hold different values, resolve to the same target data **16**.

[0008] **FIGS. 1B and 1C** illustrate one manner in which such a scenario can occur. Here, two data objects A **4** and D **16** are stored in a memory **2**A **(FIG. 1B)**, with a pointer P_1 **10** directly referencing data object D **16**. As a result of data compaction or other operations, data object D **16** is moved, as shown in memory **2**B **(FIG. 1C)**, and a new forwarding pointer **12** is inserted into data object D's old location. Thus pointer P_1 **10** now points to the forwarding pointer **12** which points to the new location of data object D **16**. Pointer P_1 **10** is therefore now an indirect pointer. Meanwhile, a new direct pointer P_2 **14** has been created which points to the new location of data object **16**, resulting in the combination of pointers pictured in **FIG. 1A**.

[0009] The presence of this aliasing necessarily introduces run time costs in order to ensure correctness of execution. In Luk, two specific problems are identified. First, direct pointer comparisons are no longer a safe operation; some mechanism must be provided for determining the final addresses of the pointers. Second, seemingly independent memory operations may no longer be reordered in out-of-order machines.

[0010] In Luk, the problem of pointer comparisons is addressed by inserting code to determine the final address for each pointer, unless the compiler is able to determine that the pointers do not point to relocated objects. The overhead of this approach is potentially large.

[0011] In the best case, both target memory words will be resident in the cache, neither of them will contain a forwarding pointer, and the pointer comparison will be slowed down by roughly an order of magnitude. However, since pointer comparisons often precede a decision to perform operations on an object, a common case will be when one or both dereferences cause a cache miss, slowing down the comparison by another order of magnitude.

[0012] The solution proposed by Luk for reordering memory operations is to use "data dependence speculation," which allows loads to execute speculatively before it is known that they are independent of any preceding stores. In an architecture that supports data dependence speculation, it is fairly easy to extend the hardware to operate correctly in the presence of forwarding pointers. Luk found that this solution is effective as incorrect speculation occurs only rarely. However, Luk assumes the presence of some fairly complex hardware. For architectures in which silicon area efficiency is a concern, a lower cost alternative is preferable.

[0013] The forwarding pointer aliasing problem is an instance of the more general challenge of determining object identity in the presence of multiple and/or changing names. This problem has been studied explicitly. See, for example, Setrag N. Khoshafian, George P. Copeland, "Object Identity", Proc. 1986 ACM Conference on Object Oriented Programming Systems, Languages and Applications, pp. 406-416, incorporated by reference herein in its entirety.

[0014] A natural solution which has appeared time and again is the use of system-wide unique object IDs or UIDs. UIDs completely solve the aliasing problem, but have two disadvantages.

[0015] First, the use of UIDs to reference objects requires an expensive translation each time an object is referenced to obtain the virtual address of the object.

[0016] Second, quite a few bits are required to ensure that there are enough UIDs for all objects and that globally unique IDs can be easily generated in a distributed computing environment. In a large system, at least sixty-four bits would likely be required in order to avoid any expensive garbage collection of UIDs and to allow each processor to allocate UIDs independently.